J2SE 5

Alcune novità introdotte da J2SE 5

- Autoboxing
- Foreach
- Enum
- Varargs
- Output formattato
- Ergonomics e performance

Autoboxing-Outboxing

Le classi wrapper

- Java non è un linguaggio completamente ad oggetti
- I tipi **primitivi** (`int`, `float`, `double`, ...) non sono oggetti
- Per supportare la gestione dei tipi primitivi uniformemente a quella degli altri oggetti, fin da Java 1 sono presenti delle classi **wrapper**

Le classi wrapper

- L'utilizzo delle classi wrapper risulta **scomodo** il più delle volte:

```
Vector v = new Vector();
int i = 10;
...
Integer iObject = new Integer(i);
v.add(iObject);
...
iObject = (Integer)v.get(0);
i = iObject.intValue();
```

Autoboxing - Outboxing

- Dal JDK 1.5 è possibile utilizzare direttamente uno **scalare** in un contesto di oggetto, lasciando che sia il **compilatore** a gestire il wrapping del tipo primitivo:

```
Vector v = new Vector(10);

for(int i=0;i<5;i++){
   v.add(i);
   v.add(((float)i)*2.5);
}

for(int i=0;i<v.size();i++){
 System.out.println(i+"="+
                v.elementAt(i));
}
```

output di esecuzione

```
0=0
1=0.0
2=1
3=2.5
4=2
5=5.0
6=3
7=7.5
8=4
9=10.0
```

foreach

foreach

- È ora supportato il ciclo **foreach** tramite una sintassi particolare di `for`:

```
for( variable_type variable_name : list )
```

- Un ciclo foreach consente di scorrere tutti gli elementi di una **struttura dati** (ad esempio un array) gestendo automaticamente:
  - l'**incremento** di eventuali indici
  - l'**assegnamento** della variabile di ciclo all'elemento successivo

foreach: esempio

```
public class foreach{
  public static void main(String argv[]){
    String array[] = new String[10];
    for(int i=0;i<array.length;i++){
   array[i] =
          new String("Stringa n."+i);
    }

  // utilizzo del foreach
    for(String s: array){
      System.out.println(s);
    }
  }
}
```

output di esecuzione

```
Stringa n.0
Stringa n.1
Stringa n.2
Stringa n.3
Stringa n.4
Stringa n.5
Stringa n.6
Stringa n.7
Stringa n.8
Stringa n.9
```

foreach: lavorare con strutture dati

```
import java.util.Vector;
public class foreach2{
    public static void main(String argv[]){
      Vector v = new Vector(10);

      for(int i=0;i<10;i++){
          v.add("String n."+i);
      }

      // uso il foreach
      // ATTENZIONE: uso di object!
      for(Object s: v){
          System.out.println(s);
      }
    }
}
```

ATTENZIONE: si deve usare un Object perché un Vector memorizza (e restituisce) Object. foreach non può effettuare i cast automaticamente! Il problema può essere risolto con generics.

Foreach e generics

- La libreria Java supporta appieno generics:

```
public static void main(String argv[]){
        Vector<String> vettore = new Vector<String>();
        for(int i=0;i<10;i++){
            vettore.add(new String("stringa n."+i));
        }
        for(String s: vettore){
            System.out.println(s);
        }
    }
```

Quando usato con generics, foreach lavora sul tipo di dato corretto!

foreach solo per estrazioni

- `foreach` non può essere usato come ciclo di assegnamento, ma solo di estrazione:

```
public static void main(String argv[]){
   String array[] = new String[10];
      for(String s: array){
      s = "CIAO";
   }
   for(String s:array){
      System.out.println(s);
   }
}
```

output di esecuzione

```
null
null
null
null
null
null
null
null
null
null
```

foreach: considerazioni

- Tutto il lavoro "sporco" viene svolto dal **compilatore**
- Non si ha accesso all'**indice** numerico della lista (non si è a conoscenza della posizione dell'elemento corrente)
- Riduce il rischio di **errori banali**, come indici fuori dai limiti o cicli annidati sulla stessa variabile
- Non è stata introdotta una parola chiave `foreach` (come in Perl, C-shell,…) per rispetto al ***legacy-code***

Preparare una classe per foreach

- Affinché una **struttura dati** (o collezione di dati) sia utilizzabile in un ciclo foreach, occorre che implementi l'interfaccia `Iterable`

java.lang

Interface Iterable<T>

Type Parameters:

`T` - the type of elements returned by the iterator

All Known Subinterfaces:

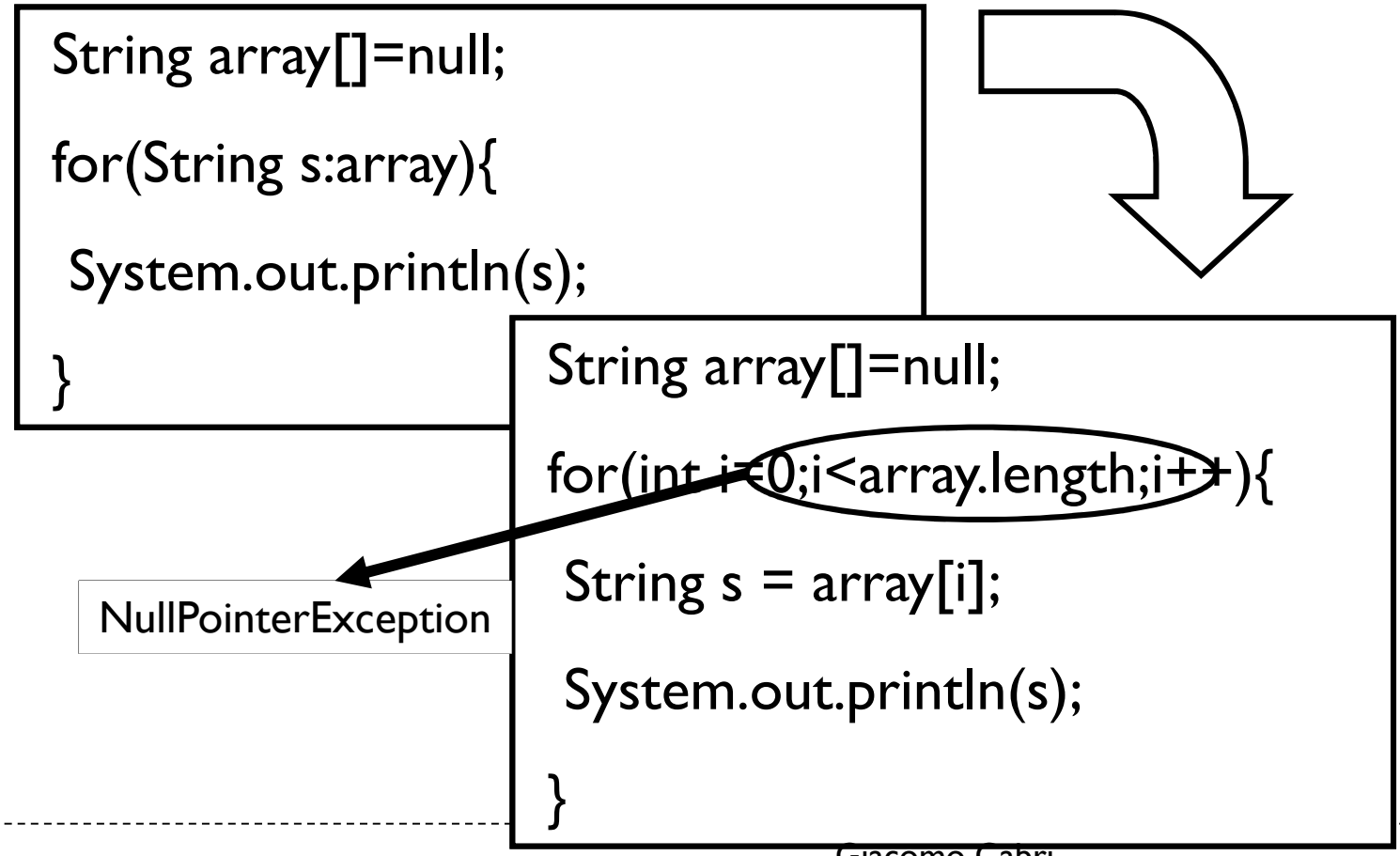
BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Collection<E>, Deque<E>, DirectoryStream<T>, List<E>, NavigableSet<E>, Path, Queue<E>, SecureDirectoryStream<T>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BatchUpdateException, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DataTruncation, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, RowSetWarning, SerialException, ServiceLoader, SQLClientInfoException, SQLDataException, SQLException, SQLFeatureNotSupportedException, SQLIntegrityConstraintViolationException, SQLInvalidAuthorizationSpecException, SQLNonTransientConnectionException, SQLNonTransientException, SQLRecoverableException, SQLSyntaxErrorException, SQLTimeoutException, SQLTransactionRollbackException, SQLTransientConnectionException, SQLTransientException, SQLWarning, Stack, SyncFactoryException, SynchronousQueue, SyncProviderException, TreeSet, Vector

foreach: una nota di demerito

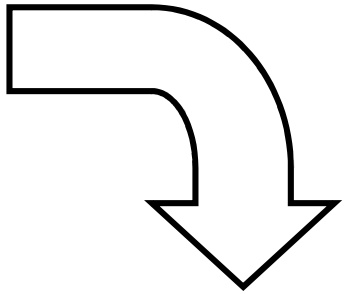
- L'implementazione di foreach **non** è *null-safe* (al contrario di Perl):



foreach: una nota di demerito

- Una implementazione più sicura sarebbe stata:

```
String arr []=null;
for(String s:arr){
 System.out.println(s);
}
```



```
String arr []=null;
for(int i=0; arr!=null && i<arr.length; i++){
 String s = array[i];
 System.out.println(s);
}
```

varargs

varargs

- È ora possibile definire metodi che accettino un numero **variabile** di argomenti
- Gli argomenti devono avere tutti lo **stesso tipo**, al limite `Object` (che equivale a "tutti i tipi possibili")
- La sintassi prevede l'uso dell'operatore `...` fra il tipo e il nome associato all'argomento
- Simile al concetto di variadic in C/C++, si pensi alla **`printf`**`(..)`

varargs: esempio

```
public class varargs {

    public void mvar(String...pars){
        for(int i=0;i<pars.length;i++){
            System.out.println("Parametro "+i+
                                        " = "+pars[i]);
        }
    }

    public static void main(String argv[]){
        varargs v = new varargs();
        v.mvar(“ALFA",“BETA",“GAMMA");
    }

}
```

L’argomento di una funzione varargs è tramutato automaticamente in un array

varargs: esempio equivalente

```
public class varargs {

    public void mvar(String...pars){
    // uso il foreach
        for(String s: pars){
            System.out.println(s);
        }
    }

    public static void main(String argv[]){
        varargs v = new varargs();
        v.mvar(“ALFA",“BETA",“GAMMA");
    }

}
```

varargs: output degli esempi

output di esecuzione

```
Parametro 0 = ALFA
Parametro 1 = BETA
Parametro 2 = GAMMA
```

Versione con ciclo for normale

Versione con ciclo foreach

output di esecuzione

```
ALFA
BETA
GAMMA
```

Overload

- È possibile **sovraccaricare** un metodo varargs con una lista di argomenti dello stesso tipo

- Verrà eseguito per **primo** il metodo il cui prototipo corrisponde perfettamente all'invocazione

Overload: esempio

```
public class varargs {
 public void mvar(String...pars){
   for(int i=0;i<pars.length;i++){
      System.out.println("Parametro "+i+" = "+pars[i]);
   }
 }

 public void mvar(Object...pars){
  for(Object o: pars){
     System.out.println("Parametro di tipo:“+o.getClass());
  }
 }

 public void mvar(String s1, String s2){
  System.out.println("stringhe "+s1+" "+s2);
 }
```

```
v.mvar(“ALFA",“BETA",“GAMMA");
v.mvar("Stringa1","Stringa2");
v.mvar(new Object(), new Integer(10));
```

Overload: array

- Siccome il compilatore tratta i metodi varargs come metodi ad array, il sovraccarico con un array viene **impedito**:

```
public class varargs {
   public void mvar(String...pars){...}

   public void mvar(String[] pars){
        for(String s:pars){
            System.out.println("String array:
  "+s);
        }
    }
}
```

output di compilazione

```
varargs.java:31: mvar(java.lang.String...) is already defined in varargs
    public void mvar(String[] pars){
```

enum

Enumerazioni

- È stata aggiunta la parola chiave `enum`, che consente di definire **enumerazioni**

- Un utilizzo classico è come lista *static* di valori

Si presti attenzione a non confondere una enumerazione con una serie di costanti. Seppur simili, una serie di costanti acquisisce importanza in base al **valore** che si attribuisce alle stesse, mentre una enumerazione ha importanza solo al fine di **distinguere** gli elementi nella lista.

Enumerazioni: esempio

```
public class EventType{
    public static enum type {
      OPEN,
      CLOSE,
      EXIT,
    };
}
```

Enumerazioni: esempio

```
public class Main1{
    // metodo che accetta un valore enumerato
    public void fireEvent(EventType.type event){
      System.out.println("L'evento ricevuto e'");

      if(event == EventType.type.OPEN)
          System.out.println("APERTURA");
      else
        if(event == EventType.type.CLOSE)
          System.out.println("CHIUSURA");
        else
        if(event == EventType.type.EXIT)
          System.out.println("USCITA");
   }
```

Enumerazioni: esempio

```
public static void main(String argv[]){
  Main1 m = new Main1();
  m.fireEvent(EventType.type.OPEN);
  m.fireEvent(EventType.type.CLOSE);

  System.out.println("Come risulta stampato? -->“

  +EventType.type.OPEN);
    }
}
```

output di esecuzione

```
L'evento ricevuto e'

APERTURA

L'evento ricevuto e'

CHIUSURA

Come risulta stampato? -->OPEN
```

Considerazioni

- Le enumerazioni sono trattate come oggetti:

```
$ ls -l
total 36
-rw-r--r--  1 luca users  306 2004-12-09 13:52 EventType.class
-rw-r--r--  1 luca users  121 2004-12-09 13:52 EventType.java
-rw-r--r--  1 luca users 1251 2004-12-09 13:52 EventType$type.class
-rw-r--r--  1 luca users 1200 2004-12-09 14:09 Main1.class
-rw-r--r--  1 luca users  629 2004-12-09 14:09 Main1.java
```

Che tipo di oggetto è un'enumerazione?

```
public void fireEvent(EventType.type event){
  ...
  // analizzo la classe dell'evento
  Class eventClass = event.getClass();
  System.out.println("Nome della classe di evento: " +
eventClass.getName());
  for(Field f: eventClass.getFields()){
    System.out.println("Campo: "+f.getName());
    System.out.println("\t visibilita': ");
    int modificatore = f.getModifiers();
    if(Modifier.isPublic(modificatore)){
      System.out.println("\tpubblico");
    }else if(Modifier.isPrivate(modificatore)){
      System.out.println("\tprivato");
    }
  }
}
```

Che tipo di oggetto è un'enumerazione?

output di esecuzione

```
Nome della classe di evento:  enumerazioni.EventType$type
Campo: OPEN
                visibilita':
               pubblico
Campo: CLOSE
                visibilita':
               pubblico
Campo: EXIT
                visibilita':
               pubblico
```

Output Formattato

L'output formattato

- Le classi di output su stream (ad es. `PrintStream`) di Java hanno il pregio di avere un utilizzo molto semplice ed immediato, ma non consentono un elevato **controllo** sull'output effettivo

- Altri linguaggi, come ad esempio il C/C++ consentono un accurato controllo dell'output mediante l'uso delle funzioni appartenenti alla famiglia **printf**

Differenze fra i metodi Java e printf

- Le differenze fra i metodi `println(..)` di un `PrintStream` Java e la `printf(..)` del C/C++ sono le seguenti:
  - `printf` è una funzione **variadic**, `println` no
  - `printf` richiede come primo argomento una stringa di **formato**, seguita dai parametri da convertire in stringa
  - `println` consente di inserire il parametro da stampare direttamente nella **posizione** in cui deve essere stampato (utilizzo dell'operatore + su stringa)
  - `printf` ha una implementazione **unica** indipendentemente dal tipo e numero di argomenti che deve gestire, `println` è **sovraccaricato** per ogni tipo di parametro differente
  - `printf` consente di specificare la **lunghezza** dei valori, `println` richiede l'utilizzo di operazioni di substringing

Il ritorno della printf

- Per sopperire alle lacune di **formattazione**, alcune classi di libreria sono state estese: `String`, `PrintWriter` e `PrintStream` hanno acquisito ciascuna i seguenti metodi (*varargs*):

```
format(String format, Object... args);
printf(String format, Object... args);
format(Locale locale, String format,
                              Object... args);
printf(Locale locale, String format,
                              Object... args);
```

Esempio: printf

```
public class Main {
    public static void main(String[] args) {
        int    v1 = 10;
        float  v2 = (float)3.69;
        String v3 = "Ciao";
        System.out.printf("valori: v1=%d v2=%f
  v3=%s\n",v1,v2,v3);

        String s = String.format("valori: v1=%d
  v2=%f v3=%s",v1,v2,v3);
        System.out.println(s);
    }
}
```

output di esecuzione

```
valori: v1=10 v2=3,690000 v3=Ciao
valori: v1=10 v2=3,690000 v3=Ciao
```

Ergonomics & performance

Ergonomics

- La JVM include ora una serie di opzioni per il **tuning** e il **profiling** delle applicazioni

- Le opzioni di ergonomics coprono:
  - La scelta del **garbage collector**
  - L'utilizzo del nuovo sistema di **class-sharing**
  - Il **dimensionamento** dell'heap e degli spazi generazionali
  - E altri aspetti

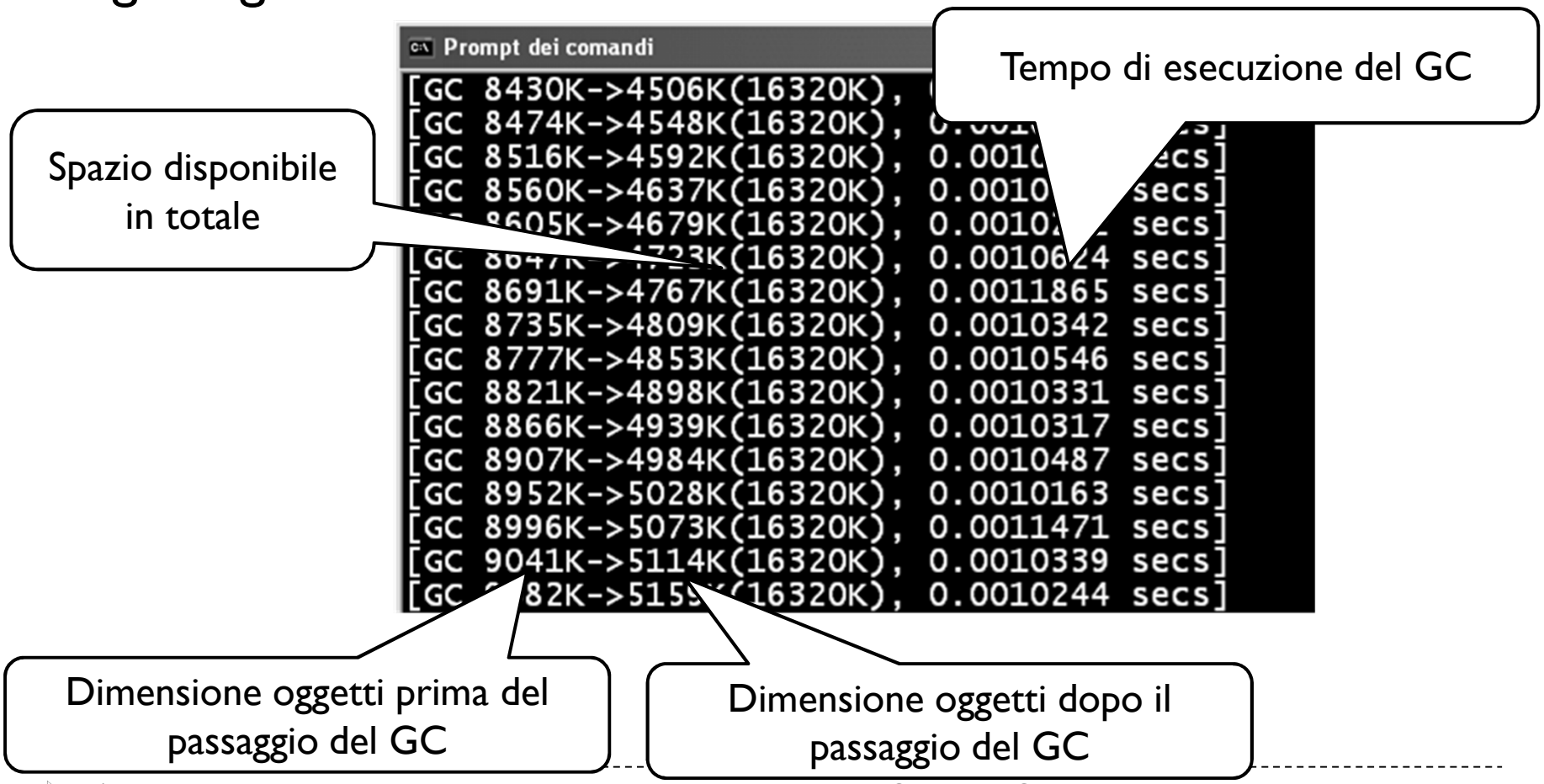
La scelta del garbage collector

Sono disponibili le seguenti opzioni:

- `-Xnoclassgc` **disabilita** la class garbage collection
- `-Xincgc` (mark & sweep) abilita il garbage collector **incrementale**
- `-XX:+UseParallelGC` abilita il GC **generazionale** parallelo (throughput collector)
- `-XX:+UseConcMarkSweepGC` versione **parallela** di `–Xincgc`

Il garbage collector in azione

- Usando il flag `-verbose:gc` è possibile osservare il garbage collector in azione:



La dimensione della memoria

- È possibile un **controllo** accurato sulla dimensione della memoria heap:
  - `-Xms` imposta la dimensione **iniziale** dell'heap
  - `-Xmx` imposta la dimensione **massima** dell'heap
  - `-Xss` imposta la dimensione dello **stack** di un thread
  - `-XX:MinHeapFreeRatio`, `-XX:MaxHeapFreeRatio` impostano il limite inferiore/superiore (in percentuale) dell'heap libero per forzare un passaggio del GC. Per un architettura a 32 bit i valori sono rispettivamente 40% e 70%

Un esempio casalingo

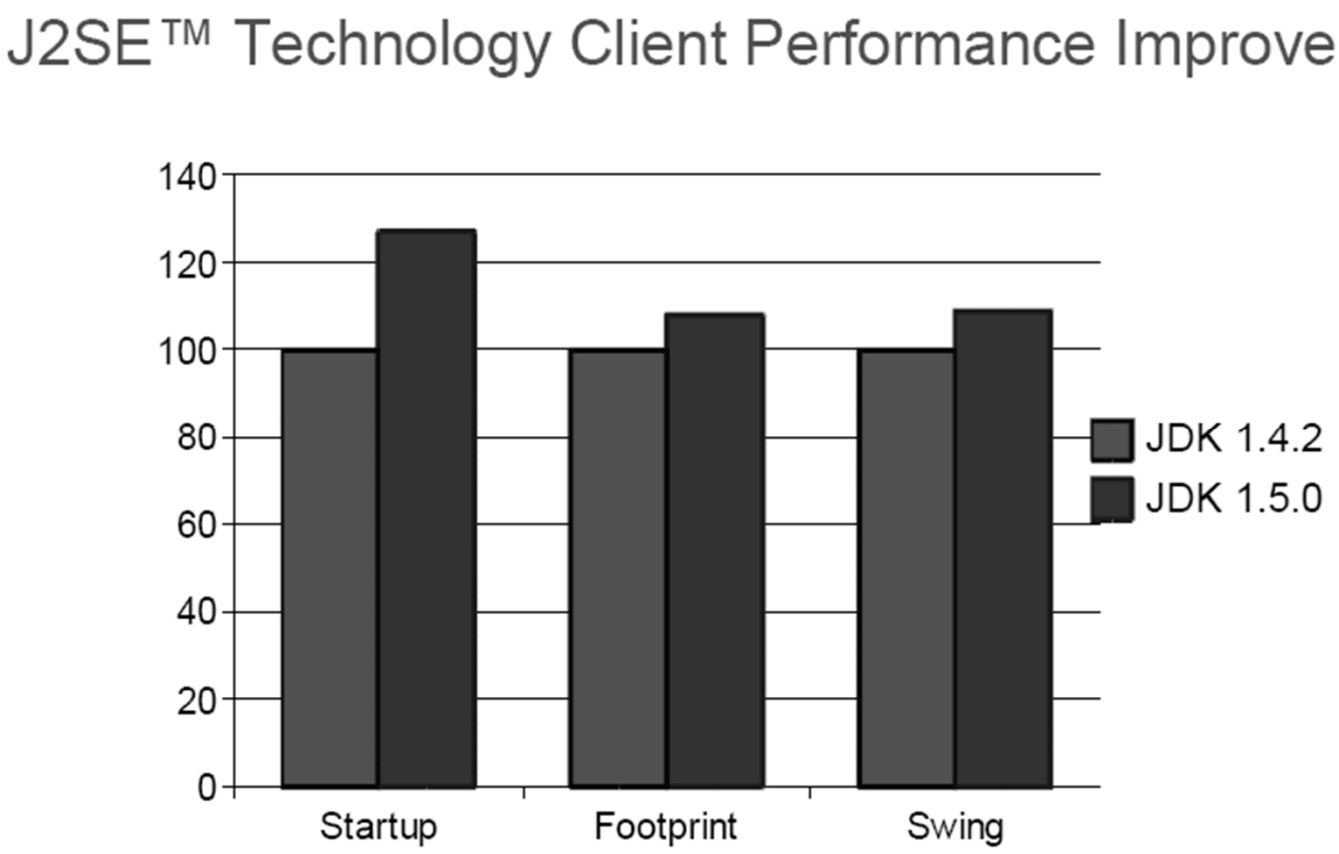
```
public class runner{
  public static void main(String argv[]){
    long start,stop;
    start = System.currentTimeMillis();
    for(int i=0;i<10000;i++){
        for(int j=0;j<10000;j++){
           String s = new String("Una stringa
                              abbastanza lunga!!!!!!");
           }
         JFrame f = new JFrame("Nuovo frame");
    }

    stop = System.currentTimeMillis();
    System.out.println("Tempo di esecuzione “
                                  +(stop-start)+" ms");
  }
}
```

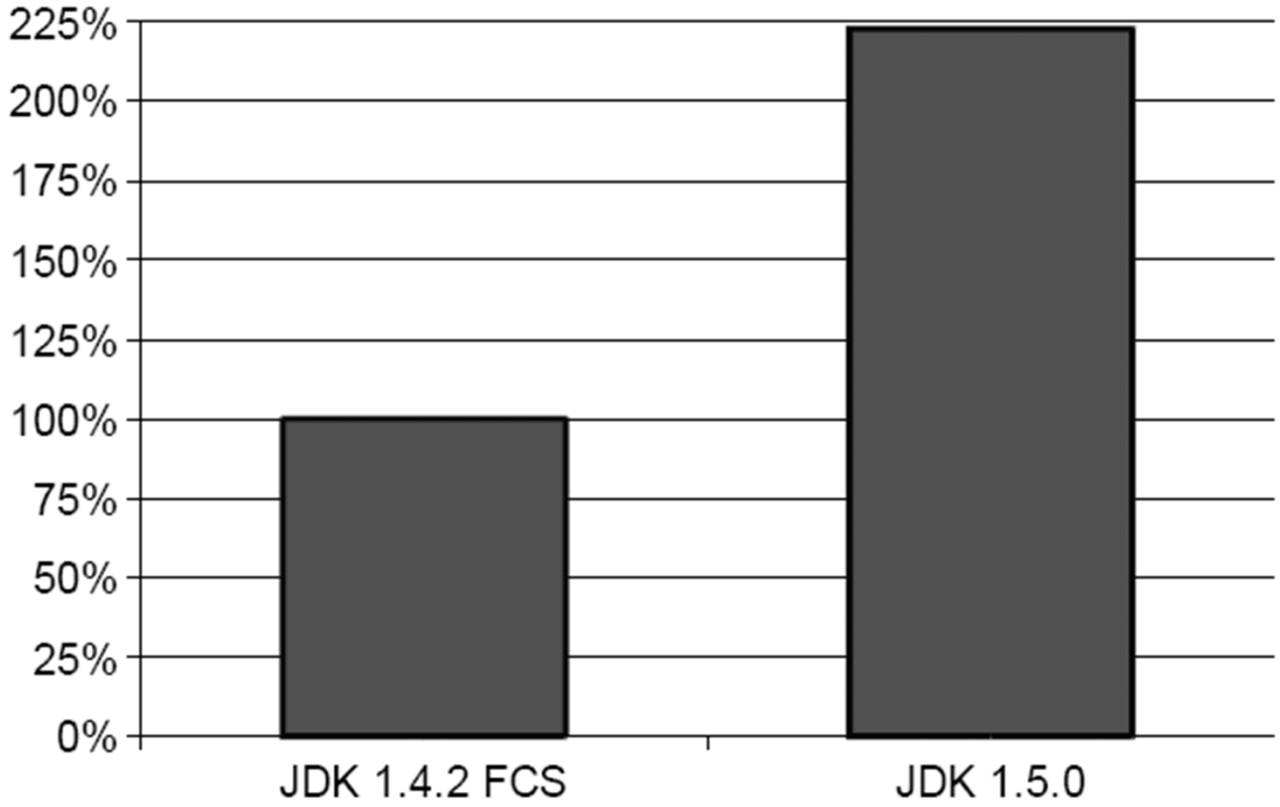
Un esempio casalingo - risultati

normale	**12518 ms**
`-noclassgc`	**12478 ms**
`-incgc`	**9584 ms**
`-int`	**47218 ms**
`-Xms 64000000`	**9033 ms**
`-Xms 64000000 –XX:MinHeapFreeRatio=1`	**8843 ms**
`-Xms 64000000 –XX:MinHeapFreeRatio=1 -XX:MaxHeapFreeRatio=99`	**8682 ms**

Prestazioni del JDK 1.5



Prestazioni del JDK 1.5



Source: Sun Microsystems
8 CPU 1.2Ghz Sun Fire v880
Solaris Next 64bit